SIMULATION OF ARITHMETIC AND BOOLEAN FUNCTIONS ON TURING MACHINES

by

RICHARD DALE CHELIKOWSKY

B. S., Kansas State University, 1961

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1962

TABLE OF CONTENTS

INTRODUCTION

A Turing machine consists of a two-way potentially infinite paper tape and a computing element. The paper tape is divided into squares which initially have zero printed on each square. The computing element has a finite number of states and is capable of reading from a particular square and overprinting a zero or one on that square. It also is capable of moving the tape one square to the right or left.

At a given time the computing element will be in a certain state and will be reading the entry on a particular square of the tape. The next operation will be determined by the current state and the symbol being read on the tape. This operation consists of overprinting a zero or one on the scanned square, shifting to the right square or left square and then assuming a new or unchanged state. A Turing machine can compute numbers by following a program which is but a collection of states. This thesis shall demonstrate numeric and Boolean algebra computations.

TOPICS RELATED TO TURING MACHINES

Computable Numbers

A. M. Turing originated the idea of a primitive computer. According to Turing's definition (10), a number is computable if its "floating-point decimal" can be written by a machine. Computable numbers also can be considered as those numbers for which an algorithm exists that can be used to compute their values. The class of computable numbers is quite large. Two particular classes are real algebraic numbers and real transcendental numbers such as π and e. A Turing machine is not allowed to print symbols

indefinitely, hence, computable irrational numbers must be approximated.

The class of computable numbers does not include all well-defined numbers. A well-defined number may not be computable because there exists no algorithm for calculating the value of the number, but yet the number has a very definite small or large value associated with it. An example of a well-defined number that is not computable is the number of squares that a given Turing machine overprints on a particular tape. Given the program and the input tape of a Turing machine there is no algorithm for determining the number of squares that will be overprinted. The tape must be run through the given Turing machine and the overprinted squares counted if the answer is to be found.

Using Turing's definition of computability, it can be shown (1) that any number which is defined by a general recursive function is a computable number. Because of this it is possible to tell if a number is computable simply by noting recursiveness.

Turing Machines and Digital Computers

Turing's original work with computable numbers is considered to be fundamental in the development of a theory of digital computers. The structure of Turing machines and digital computers implies deterministic computations in the sense that, while either of them is in operation the entire future is specified by its present status by means of the program. It follows that all ordinary digital computers which do not contain random or probabilistic elements are equivalent to some Turing machine.

Because of the Turing machine's structural simplicity, the great problem and focus of attention is the program. This contrasts with present day digital computers whose greater structural complexity enables

use of simple programs such as FORTRAN. In this way, trade-off between computer complexity and programming complexity is brought to light.

The Decision Problem

One of the primary tasks of present day mathematicians is that of determining whether various propositions concerning mathematical objects are true or false. Problems which inquire as to the existence of an algorithm for deciding the truth or falsity of a whole class of statements, as opposed to a question concerning a single proposition, are known as decision problems. A positive solution for a decision problem consists of giving an algorithm for solving the problem, while a negative solution consists of showing that no solution exists.

Gödel and Turing on the Hilbert Entscheidungsproblem

Basically the Entscheidungsproblem (decision problem in German) was to find a general process for determining whether a given formula of the functional calculus K is provable. In the 1920's, a great number of mathematicians were concerned with finding a solution to the Entscheidungsproblem. Finally, in 1931 Gödel (2) showed that there were propositions U such that neither U nor -U is provable. This in itself was not a solution to the Entscheidungsproblem. However, it was a very definite result which set the basis for the eventual solution of the problem by Turing.

In 1936, A. M. Turing (10) published a now classic paper: "On Computable Numbers with an Application to the Entscheidungsproblem." Here, he developed the notion of a computing machine and then used it to show that the Entscheidungsproblem has no solution. He showed that there can

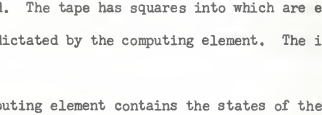
be no machine which, supplied with any formula of the functional calculus K will eventually say whether the particular formula is provable in K.

THE NATURE OF TURING MACHINES

The Tape and the Computer Components of a Turing Machine

The paper tape which moves through the computing element is potentially infinite in length. This means that if the tape needs to be longer more tape is available. Thus a Turing machine can store as much information as may be needed. The tape has squares into which are entered the symbols zero or one dictated by the computing element. The initial entries are zeros.

The computing element contains the states of the Turing machine. These can be conveniently represented by cards which tell exactly which operation is to be performed. The k-th state of a Turing machine with n states has the following representation on a card.



P_o, P_1, S_o, and S_1 all can assume values of zero and one, while both C_o and C_1 can be any number from zero to n.

Assume that a Turing machine is in state k and that the computer is scanning a square which has a zero printed upon it. Card k's instructions are to overprint the zero with P_o (either a zero or a one) and shift the tape one square so that the machine scans the square immediately on the

right of the one it had scanned previously if S_o is a one or just to the left of the square it was scanning if S_o is a zero. The final instruction of state k is for the machine to assume state C_o. Had the original scanned square contained a one instead of a zero the instructions on the one line would then have been followed.

A Turing machine starts its operation by performing the instructions of card one on the original square it was scanning and then assuming different states until it receives an instruction to assume state zero, the stop state. There is no stipulation as to the number of times a machine can be in a particular state or to the order in which the machine passes through the states.

Number Representation on a Turing Machine

There are several codes which can be used to represent numbers on a Turing machine. Turing used a binary representation. A positive integer n can also be represented on the tape of a Turing machine by a string of ones in n + 1 adjacent squares.

The second representation is more frequently used because of that Turing machine's lesser structural complexity. Another advantage of the second code is that the absence of a signal will not be interpreted as an order. Hence, the integer zero is represented as a single one on that machine's tape.

Negative integers are not representable on a Turing machine. This does not impose any serious restrictions, since most mathematical operations can be expressed in terms of positive integers.

Variants on Turing Machines

The literature contains a large number of papers on Turing machines. Most authors consider Turing machines from different viewpoints and use many different representations for Turing machines. There are two main ways to consider Turing machines. One way is to define a Turing machine as a finite set of quintuples $q_i S_j S_k D q_n$. This method follows Turing (10) and corresponds to the definition previously discussed. The q_i refers to the state of the machine. The S_j refers to the scanned symbol and the S_k to the symbol printed. The D refers to shifting right or left one square or not moving at all. The state the machine will go into next is given by q_n.

The other way of defining a Turing machine follows Davis (1). Davis' formulation considers a Turing machine as a finite set of quadruples. Davis' definition differs from Turing's in that Davis does not consider the possibility of both overprinting a symbol and shifting one square in the same operation.

All three of the definitions described are equivalent to each other. The definition of a Turing machine presented by the writer is the invention of Tibor Rado of Ohio State University. Rado's method of treating Turing machines is the clearest and most straight forward of any available in the literature.

The main difference between Turing's approach and Rado's is that Turing considered the reading and printing of symbols other than zero or one. Turing also allowed the machine to scan the same square after it had printed a symbol upon that square.

Universal Turing Machines

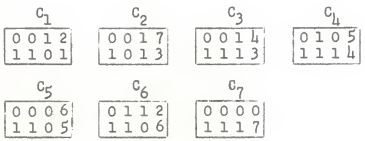
Turing also developed the concept of a universal computing machine. This machine has the property that if a suitably coded description of any Turing machine is printed on its input tape, it will act like the machine described provided the machine is started at a suitable point and in a suitable state. The machines act in the same manner because the universal machine will compute the same number as the described machine, but normally at a much slower rate. Universal Turing machines generally use more than two symbols (say, zero and one) and are quite complex in their structure. At the present time the smallest universal machine uses five symbols and has eight states. The state-symbol product is representative of the size of a universal Turing machine. This universal machine with a product of forty was constructed by Watanabe (12).

ELEMENTARY PROGRAMS

The Copying Turing Machine

The simplest recursive function of x is $f(x)=x$ and its determination generates the copying machine. A copying machine will copy any string of ones on the input tape to the right of the given string with one separating zero between the two strings. The scanning should start at the rightmost one of the string of ones when the program begins. This is the standard starting position of a Turing machine. When the machine has copied the string it will stop at the rightmost one of the copied string. This is illustrated as follows:

The program that enables a Turing machine to copy strings of ones consists of seven cards:



The program is called COPYR for "copy right".

Programming the Turing Machine

The difficult problem of programming a Turing machine can be alleviated by subprograms. A flow chart of subprograms similar to that used by digital computer programmers may be constructed which shows the entire operation of the Turing machine. The price that is paid for this clear picture of the Turing machine's operation is the addition of more states to the machine. The time of execution of the program is related to the number of states and the input tape.

The seven card COPYR program shown previously is a good example of a Turing machine which takes fewer cards than the equivalent program obtained by assembling subprograms. In this case the larger program requires twenty-nine cards, but the operation of the Turing machine is much clearer than the seven card program. Reduction of states is a matter of much additional

insight. The minimum number of states is an example of a small uncomputable number.

Basic Machines or Subprograms

There are ten basic machines or subprograms from which larger programs can be constructed. These ten subprograms and their designations are shown below:

1) PZ (Print a zero on the scanned square.)

	C_1	C_2
0	0 1 2	0 0 0 0
1	0 1 2	1 1 0 0

2) PONE (Print a one on the scanned square.)

	C_1	C_2
0	1 1 2	0 0 0 0
1	1 1 2	1 1 0 0

3) FZR (Find the first zero to the right and stop there.)

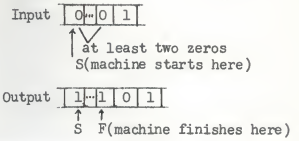
	C_1	C_2	C_3
0	0 1 2	0 1 3	0 0 0 0
1	1 1 2	1 1 2	1 1 0 0

4) FONER (Find the first one to the right and stop there.)

	C_1	C_2	C_3
0	0 1 2	0 1 2	0 0 0 0
1	1 1 2	1 1 3	1 1 0 0

5) RSTRR (Restore the string to the right.)

C_1	C_2
0 1 1 1	0 - - -
1 1 0 2	1 0 0 0



6) FZZR (Find the first pair of zeros to the right and stop under the rightmost zero.)

C_1	C_2	C_3	C_4
0 0 1 2	0 0 1 3	0 0 1 4	0 0 0 0
1 1 1 2	1 1 1 2	1 1 1 2	1 1 0 0

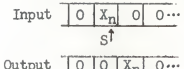
7) GOTOR (Shift one square to the right.)

C_1
0 0 1 0
1 1 1 0

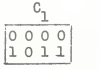
8) X_n is a sequence of zeros and ones which does not contain two zeros in a row. The n refers to the number of strings of ones in the sequence. Two adjacent zeros signify a barrier or end of sequence.

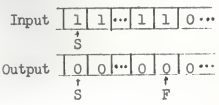
ITOR (Shift X_n right one square; stop at the right end of X_n.)

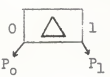
C_1	C_2
0 0 0 0	0 1 1 1
1 0 1 2	1 1 1 2



9) MOPR (Erase all ones in the string to the right and stop at the last one.)

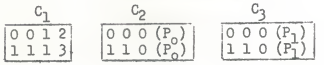




10) 

(Branching Instruction)

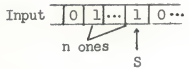
If a one is seen go to the first card of subprogram P_1. If a zero is seen go to the first card of subprogram P_o.

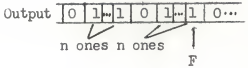


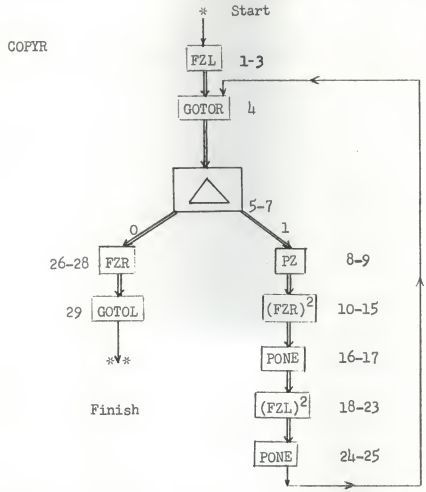
Subprograms two through nine may be modified by replacing all zeros and ones in the shift column of the cards by ones and zeros respectively. The subprograms will then work to the left instead of the right. FZR, for example, becomes FZL which finds the first zero to the left.

Flow Chart of the Copying Machine

We can now discuss the construction of the program from the ten basic programs.







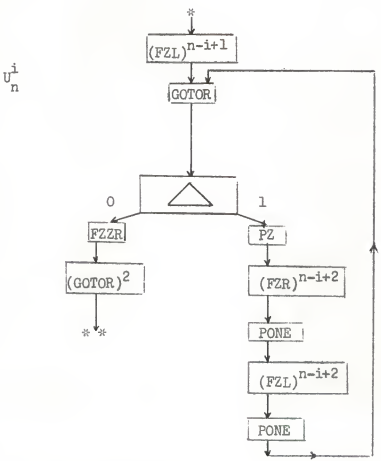
Starting with the first instruction, FZL, the subprograms are followed until the last instruction, GOTOL, is executed. Instructions of a subprogram with a superscript k should be followed k times. The numbers to the right of the boxes indicate the card numbers of the particular subprogram. The double arrows aid the assignment of card numbers to the subprograms, because they link non-looping subprograms. The card numbers are assigned to subprograms in consecutive order until a single arrow is reached. Card numbering will continue with the next uncounted subprogram.

Flow Chart of the Identity Function U_n^i

Input | 0 | X_n | 0 | 0⋯ |
↑S

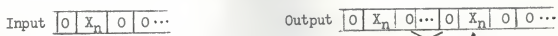
Output | 0 | X_n | 0 | x_i | 0 | 0⋯ |
↑F

X_n is any sequence of zeros and ones which does not contain two zeros in a row. The subscript n refers to the number of strings of ones in the sequence. The i-th string of ones in X_n is x_i. The program can be said to compute $U_n^i(x_1, x_2, \cdots x_i, \cdots x_n) = x_i$. When n=i=1 the program is equivalent to COPYR.

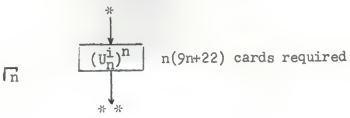


This program takes 9(n-i)+31 cards.

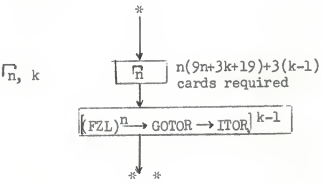
Flow Chart of the Function $\lceil n, k$



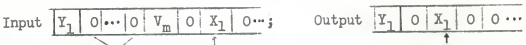
When k=1 the function $\lceil n,k$ is written $\lceil n$ and its program is



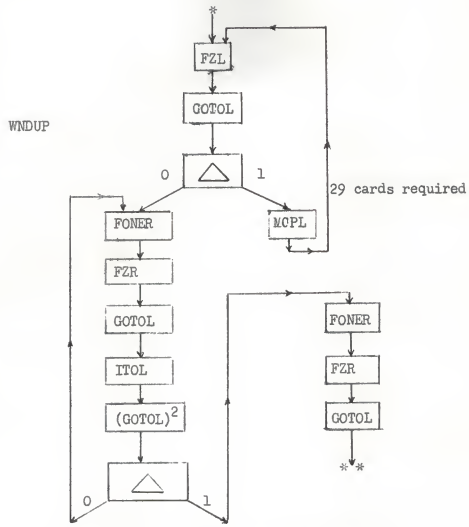
In general when k≥1 the program for $\lceil n,k$ is



Flow Chart for the WNDUP Program



The WNDUP program erases the sequence V_m and moves X_1 over within one separating zero of Y_1. A subscript on an upper case letter (such as X,Y,V) represents the number of strings of ones (separated by one zero) contained in the sequence of the upper case letter. V_m generally represents "scratch work" and X_1 represents the desired answer of some operation.



ARITHMETIC OPERATIONS ON TURING MACHINES

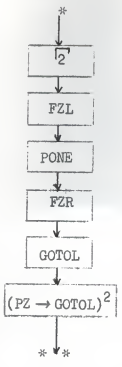
Flow Chart for Addition

Input | 0 | x | 0 | y | 0 | 0 ··· ; (S)

Output | 0 | x | 0 | y | 0 | (x+y) | 0 | 0 ··· (F)

Lower case letters on the tape represent positive integers.

ADDITION



95 cards required

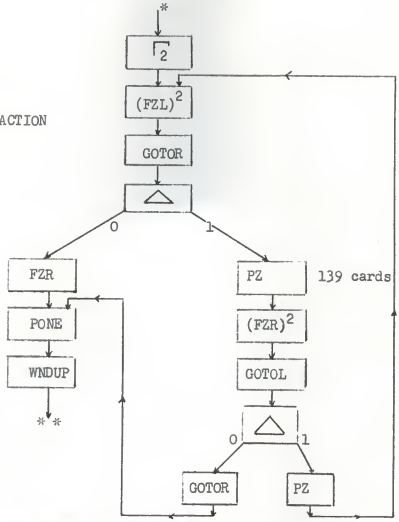
Flow Chart for Proper Subtraction

Proper subtraction is defined as follows:

$$x \div y = \begin{cases} 0 \text{ if } x < y \\ x-y \text{ if } x \geq y \end{cases}$$

Since negative integers are not representable on a Turing machine, this program finds frequent usage. The absolute value of the difference of two integers is given by the relation: $|x-y| = (x \div y)+(y \div x)$

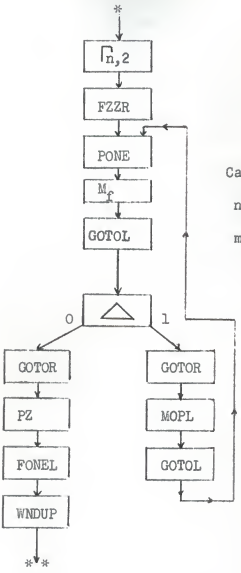
PROPER SUBTRACTION



Flow Chart for Minimalization Program

This program will compute $f(X_n)=(\mu y)\left[F(X_n,y)=0\right]$ given M_f (program to compute $F(X_n,y)$). $f(X_n)$ is the minimum value of y that satisfies the equation $F(X_n,y)=0$. This program can be used to determine arithmetic (bracket) square roots and quotients.

MINIMALIZATION

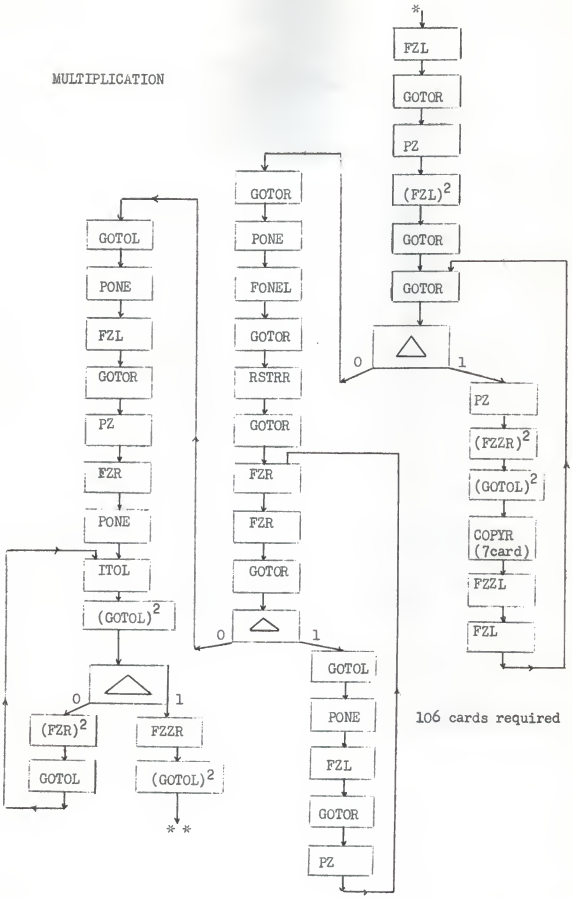


Cards required:

n(9n+25)+50+ cards of machine f.

Flow Chart for Multiplication Program

MULTIPLICATION



106 cards required

General Recursive Functions

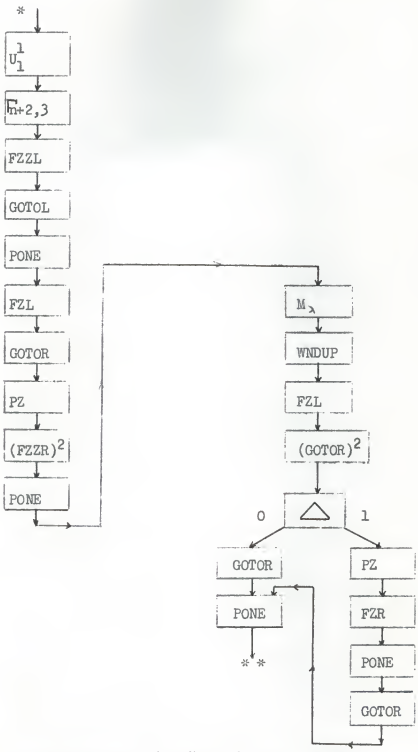
Because a Turing machine can compute any general recursive function it is of interest to construct a machine which can compute such functions. A general recursive function is described by the following relations:

$$f(X_n,0)=\lambda(X_n) \qquad f(X_n,y+1)=\Lambda\left[X_n,y,f(X_n,y)\right]$$

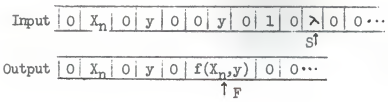
The problem is to find $f(X_n,y)$ given the machine to compute $\lambda(M_\lambda)$, the machine to compute $\Lambda(M_\Lambda)$, and y.

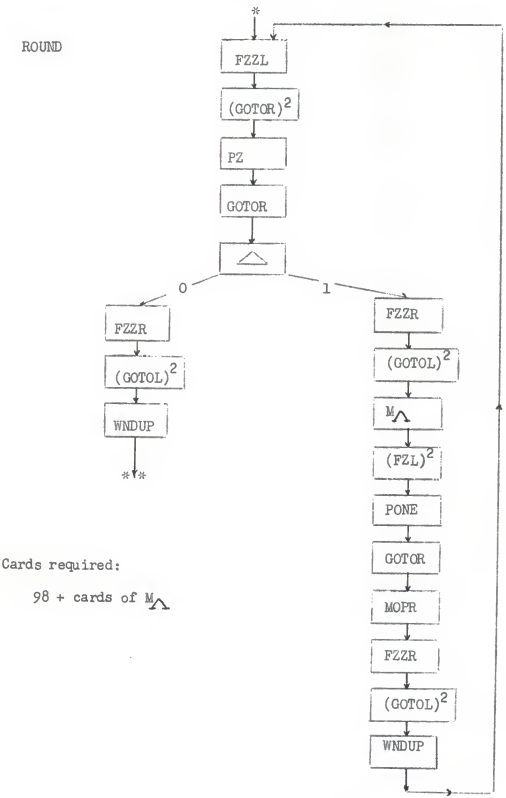
The machine is composed of two major subprograms. The first is PREP. When given the input $\boxed{0}\boxed{X_n}\boxed{0}\boxed{y}\boxed{0}\boxed{0}\cdots$ (S) the corresponding output tape is $\boxed{0}\boxed{X_n}\boxed{0}\boxed{y}\boxed{0}\boxed{0}\boxed{y}\boxed{0}\boxed{1}\boxed{0}\boxed{\lambda}\boxed{0}\boxed{0}\cdots$ (F) which is used as the input tape for ROUND the other half of the program.

PREP



Cards required: $n(9n+64)+176+$ cards of M_λ





Cards required:

98 + cards of $M_{\wedge}$

Finally, the machine GERF (general recursive function) can be formed.

Cards required:

$n(9n+64)+274+$ cards of

$M_{\wedge}$ and $M_{\curlywedge}$

BOOLEAN FUNCTIONS ON TURING MACHINES

Finite Automata, Restricted Turing Machines

Turing machines which are not allowed to shift to the left form a very special subclass of Turing machines. Such restricted Turing machines are called finite automata (4). Generally speaking, Turing machines have two main features: their ability to make decisions and their access to an essentially unlimited memory. By imposing the restriction of not shifting to the left a Turing machine is reduced to a machine with no memory.

The loss of memory is quite a severe restriction. Because of their limited memory, rather simple tasks lie beyond the reach of finite automata. For instance, there is no finite automaton which will perform in the manner of the COPYR Turing machine. This excludes finite automatons with an infinite number of states as implied in the name finite automata.

Because finite automata cannot scan any square on the tape more than once they require a prepared input tape which is not initially filled with zeros. A finite automaton with zero initial conditions is not capable of being in a particular state more than once. If allowed to return to the same state the automaton will never stop printing symbols. Such an initial

state automaton is of little interest.

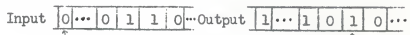
Automata Flow Charts and Subprograms

Subprograms and flow charts for finite automata can be constructed in the same manner as unrestricted Turing machines. The finite automata subprograms are as follows:

1) APZ (Print a zero on the scanned square and shift right.

2) APONE (Print a one on the scanned square and shift right.)

3) ARSTRR (Restore the string to the right.)



4) AFZR (Stop to the right of the first zero seen.)

5) AFONER (Stop to the right of the first one seen.)

6) AFZZR (Stop to the right of the first pair of zeros seen.)

C_1	C_2
0 0 2 1 1 1	0 0 0 1 1 1

7) AGOTOR (Shift one square to the right.)

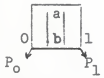
C_1
0 0 0 1 1 0

8) AITOR (Shift X_n right one square; stop two squares to the right of the last one of X_n.)

C_1	C_2
0 0 0 1 0 2	0 1 1 1 1 2

9) AMOPR (Erase all ones in the string; stop two squares to the right of the last one.)

C_1
0 0 0 1 0 1

10) 

(Branching instruction)

If a zero is seen, overprint with an "a", shift right and go to the first card of subprogram P_o. If a one is seen, overprint with a "b", shift right and go to the first card of subprogram P_1.

C_1
0 a P_o 1 b P_1

An inspection of the automata subprograms shows that each subprogram is analogous to an unrestricted arithmetic Turing machine subprogram. Some of these analogous programs are not used very frequently however.

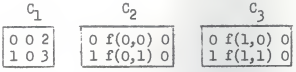
Boolean Functions and Automata

Since finite automata are limited in their computational ability, they can only perform rather specific operations. An automaton can be constructed which will compute the value of any Boolean function given the functional values corresponding to all possible input variables. Automata which evaluate 2^n-valued Boolean functions make use of the fact that it is possible to represent the elements of a 2^n-valued Boolean function by binary numbers.

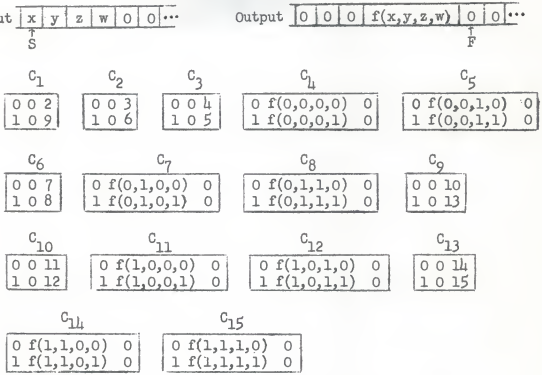
Automaton that Computes Two-valued Boolean Functions

Since finite automata always move to the right, they start at the leftmost one of the input tape instead of the rightmost one. Another difference between automata and unrestricted Turing machines is that the Boolean algebra elements (say, zero and one) are not represented on a tape in the usual integer notation of a Turing machine. The element zero is represented by a zero on the tape and the element one by a single one. Also, the shift column is omitted from the state cards because of the right moving property of finite automata.

For two literals x and y the Boolean function automaton will perform in the following manner:



As another example consider a 4 literal (x,y,z,w) Boolean function automaton.



In general the n literal case will take 2^n-1 cards, but some of the cards with the functional values will be the same for four or more literals thus permitting a reduction of cards to a maximum value of $(2^n-1)-(2^{n-1})+4 = 2^{n-1}+3$.

Binary and Gray Codes

The binary number system provides one method of representing numbers with electronic circuits which only recognize two voltage levels. Consequently, digital computers ordinarily use a binary-coded decimal

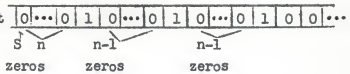
representation or a weighted binary-coded representation.

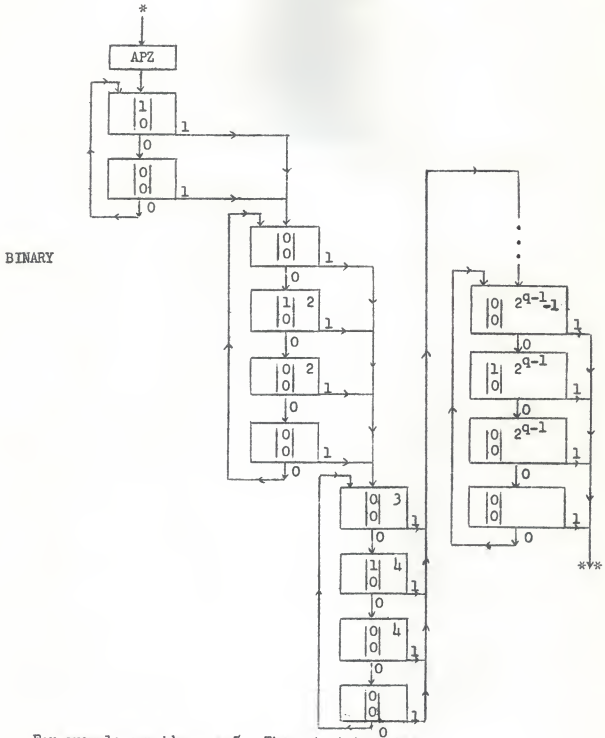
Another code frequently used by computers is known as the Gray (7) (or reflected) code. It is a method of counting in such a way that only one bit changes at a time. For devices which measure angular displacements such as shaft encoders, this is a very desirable property. For in the case of an ambiguous reading the error in resolution will not exceed the value of the least significant bit.

Because of the separate advantages of the Gray and binary codes it frequently is desirable to use both of the codes. If this is the case a conversion technique will be needed to relate the codes.

Generation of the First n Binary Numbers

The program to generate the first n binary numbers requires $3(2^q-1)$ cards. The number of ones on the tape initially is q (q = 2,3,4,$\cdots$). The integer n is related to q by the expression; min $2^q \geq n$.





For example consider n = 5. The output tape is:

output | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | ...

F

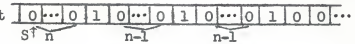
If the output tape is cut at the end of every n-th square starting with the first square at the left and the resulting q sections placed on top of each other (with the leftmost section on top) the first n binary numbers will be at hand. If this is done for the example (n = 5) the result will be:

	0	1	2	3	4
First section —	0	1	0	1	0
Second section —	0	0	1	1	0
Third section —	0	0	0	0	1

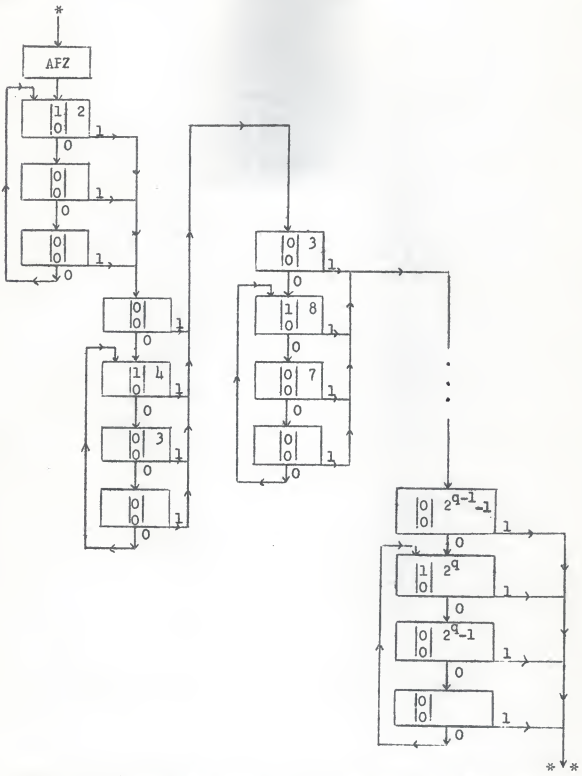
The binary numbers corresponding to 0, 1, 2, 3 and 4 are in a vertical position and are read from the bottom up with the least significant bit at the top.

Generation of the First n Gray Numbers

The program generating the first n Gray numbers is very similar to the program used to generate the binary numbers. However, this program takes $5 \cdot 2^q - (4+q)$ cards. The integer q which is the number of ones on the tape initially is related to n by the expression: $\min 2^q \geq n$.



GRAY



If the first 5 Gray numbers were desired the output tape would be:

Output 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 ...

F

The output tape shown is then cut into q sections of length n squares starting with the first five squares at the left end. The result of placing these sections on top of each other with the leftmost section on top is shown below.

	0	1	2	3	4
First section—	0	1	1	0	0
Second section—	0	0	1	1	1
Third section—	0	0	0	0	1

The Gray numbers corresponding to the integers 0, 1, 2, 3 and 4 are read from the bottom up with the least significant bit at the top.

Binary Numbers and 2^n-valued Boolean Algebras

The fact that the first 2^n binary numbers form a Boolean algebra can be used in the construction of an automaton that will compute 2^n-valued Boolean functions. A 2^n-valued Boolean algebra will exist for any set of 2^n distinct elements which satisfies the following five postulates. There are two operations in any Boolean algebra: $\cap$ (cap) and $\cup$ (cup).

Boolean Algebra Postulates.

1) The $\cup$ and $\cap$ operations are closed. Given any two elements of the set, say a and b, $a \cup b = c$ and $a \cap b = d$ where c and d are also members of the set.

2) The $\cup$ and $\cap$ operations are commutative. If this is true then $a \cup b = b \cup a$ and $a \cap b = b \cap a$.

3) There exists an identity element 0 for the $\cup$ operation and an identity element 1 for the $\cap$ operation such that $a \cup 0 = 0 \cup a = a$ and $a \cap 1 = 1 \cap a = a$.

4) The $\cup$ operation must distribute over the $\cap$ operation and the $\cap$ operation must distribute over the $\cup$ operation such that $a \cup (b \cap c) = (a \cup b) \cap (a \cup c)$ and $a \cap (b \cup c) = (a \cap b) \cup (a \cap c)$.

5) For each element b there must exist an element b' such that $b \cup b' = 1$ and $b \cap b' = 0$.

The fact that the elements 0 and 1 can have cup and cap operation tables which form a Boolean algebra can be used to show that the first 2^n binary numbers form a Boolean algebra.

Operation tables for the elements 0 and 1.

$\cup$	0	1
0	0	1
1	1	1

$\cap$	0	1
0	0	0
1	0	1

Consider two binary numbers of the same length (say, abcd and efgh). Let these two numbers be represented by A = (a,b,c,d) and B = (e,f,g,h). The cup and cap operations between these two binary numbers are defined as follows:

$A \cup B = (a \cup e,\ b \cup f,\ c \cup g,\ d \cup h) = C$

$A \cap B = (a \cap e,\ b \cap f,\ c \cap g,\ d \cap h) = D$

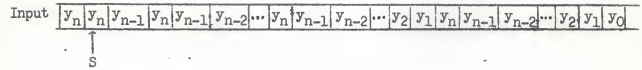
The first 2^n binary numbers contain all possible sequences of elements zero and one which are of length n. Consequently the postulate of closure is satisfied because of closure of the two elements zero and one which are involved in every cup and cap operation between binary numbers. All of the other postulates are satisfied by this same reasoning. The cup identity is the binary number zero and the cap identity is the binary number 2^n-1.

BOOLEAN TRANSFORMATIONS

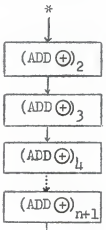
Conversion of Gray Code to Binary Code

The binary number $(x_n x_{n-1} \cdots x_1 x_0)$ equivalent (8) of a Gray number $(y_n y_{n-1} \cdots y_1 y_0)$ is $x_n = y_n$, $x_{n-1} = y_n \oplus y_{n-1}$, $x_{n-2} = y_n \oplus y_{n-1} \oplus y_{n-2}, \cdots$ $x_2 = y_n \oplus y_{n-1} \oplus y_{n-2} \oplus \cdots \oplus y_2$, $x_1 = y_n \oplus y_{n-1} \oplus y_{n-2} \oplus \cdots \oplus y_2 \oplus y_1$, $x_0 = y_n \oplus y_{n-1} \oplus \cdots \oplus y_3 \oplus y_2 \oplus y_1 \oplus y_0$.

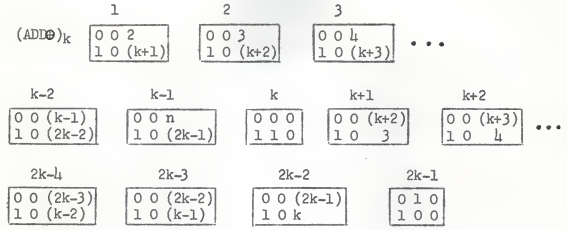
The given Gray number is entered on the tape as shown below.



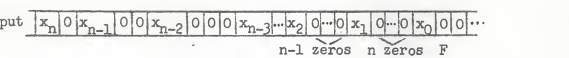
The tape is run through the program



The symbol $\oplus$ signifies addition modulo two. The subscript of $(\text{ADD}\oplus)_k$ refers to the number of digits being added modulo two. The number of cards for $(\text{ADD}\oplus)_k$ is $2k-1$.



The output tape has the following appearance.



The digit x_i is found on the $1/2(n-i+2)$ $(n-i+1)$th square from the first square on the left. The number of cards needed for the entire program is $\sum_{k=2}^{n+1} (2k-1) = n(n+2)$.

Conversion of Binary Code to Gray Code

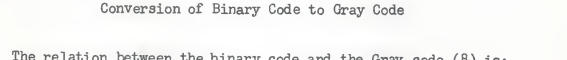
The relation between the binary code and the Gray code (8) is:

if the number $x_n \cdots x_3\ x_2\ x_1\ x_0$ (given binary number) is equivalent to the number $y_n \cdots y_3\ y_2\ y_1\ y_0$ (Gray representation of given binary number)

then $y_n = x_n$, $y_{n-1} = x_n \oplus x_{n-1}$, $\cdots$, $y_{i-1} = x_i \oplus x_{i-1}, \cdots$,

$y_0 = x_1 \oplus x_0$.

The given binary number is entered onto the tape in the following manner.



This tape is then run through the $(\text{ADD}\oplus)_2^{n+1}$ program. The resulting

output is:

Output | 0 | y_n | 0 | y_{n-1} | 0 ··· | y_{i-1} | ·· | y_0 | 0 | 0 | ···
F

The Gray digits occupy every other square of the output tape. This program requires 3(n+1) cards.

ARITHMETIC OPERATIONS WITH OTHER CODES

Binary Addition

Consider the sum of these two n-1 bit binary numbers.

$$\begin{array}{l} x_{n;0}\ x_{n-1,0}\cdots x_{4,0}\ x_{3,0}\ x_{2,0}\ x_{1,0}\ x_{0,0} \\ +\ x_{n,1}\ x_{n-1,1}\cdots x_{4,1}\ x_{3,1}\ x_{2,1}\ x_{1,1}\ x_{0,1} \qquad x_{n,0} = x_{n,1} = 0 \\ \hline z_n\ \ z_{n-1}\ \cdots\ z_4\ \ z_3\ \ z_2\ \ z_1\ \ z_0 \end{array}$$

Finite automata may be used in determining this sum. Enter the two binary numbers onto the input tape in an alternating fashion.

Input | $x_{0,0}$ | $x_{0,1}$ | $x_{1,0}$ | $x_{1,1}$ | $x_{2,0}$ | $x_{2,1}$ | ··· | $x_{n-1,0}$ | $x_{n-1,1}$ | 0 | 0 | ···
S

This input tape is run through the program $(\text{ADD}\oplus)_2^n$.

$(\text{ADD}\oplus)_2$

C_1	C_2	C_3
0 0 3	0 1 0	0 0 0
1 0 2	1 0 0	1 1 0

resulting in

Output | 0 | $x_{0,0}\oplus x_{0,1}$ | 0 | $x_{1,0}\oplus x_{1,1}$ | 0 ··· 0 | $x_{n-1,0}\oplus x_{n-1,1}$ | 0 | 0 | ···
F

The original input is run through another program: $(\text{MULT})^n$

MULT

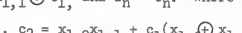
C_1	C_2	C_3
0 0 3	0 0 0	0 0 0
1 0 2	1 1 0	1 0 0

The output is

The nature of binary addition is such that $z_0 = x_{0,0} \oplus x_{0,1} \oplus c_0$, $z_1 = x_{1,0} \oplus x_{1,1} \oplus c_1$, and $z_n = c_n$. Where the c parameters are: $c_0 = 0$, $c_1 = x_{0,0}x_{0,1}$, $c_2 = x_{1,0}x_{1,1} + c_1(x_{1,0} \oplus x_{1,1})$, and $c_n = x_{n-1,0}x_{n-1,1} + c_{n-1}(x_{n-1,0} \oplus x_{n-1,1})$.

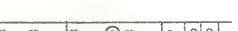
The parameters c_0 and c_1 have already been determined. The other c's are found by the following procedure.

To determine c_2 run the input tape below through the following program C.

Input 

Program C:

C_1	C_2	C_3	C_4	C_5	C_6
0 0 2 1 0 5	0 0 3 1 0 4	0 0 0 1 0 0	0 0 0 1 1 0	0 0 6 1 0 6	0 1 0 1 1 0

The output is 

The parameter c_2 is now recorded and $x_{2,0}x_{2,1}$ and $x_{2,0} \oplus x_{2,1}$ are now put on this output tape and then run through program C. This determines c_3.

New Input 

Output

This procedure is followed until all of the c parameters have been determined.

A tape may now be prepared which when run through the $(\text{ADD} \oplus)^n$ program will yield the sum of the two binary numbers.

Input $\underset{S}{|c_0|x_{0,0}\oplus x_{0,1}|c_1|x_{1,0}\oplus x_{1,1}|\cdots|c_{n-1}|x_{n-1,0}\oplus x_{n-1,1}|0|c_n|0|0|}\cdots$

Output $\underset{F}{|0|z_0|0|z_1|0|z_2|0|\cdots|0|z_n|0|0|}\cdots$

Note that $z_0 = c_0 \oplus x_{0,0} \oplus x_{0,1}$, ; $z_{n-1} = c_{n-1} \oplus x_{n-1,0} \oplus x_{n-1,1}$ and $z_n = c_n$. The z's are the digits of the binary answer written as $z_n\ z_{n-1}\cdots z_3\ z_2\ z_1\ z_0$. This program requires 6(n+1) cards and is completed in n+2 passes of the automaton.

Gray Code Addition

Lucal (5) modified the Gray code to find the sum of two Gray numbers. Consider addition of the following two modified n-1 bit Gray numbers:

$$\begin{array}{l} y_{n,0}y_{n-1,0}\cdots y_{1,0}\ y_{0,0} \\ +y_{n,1}y_{n-1,1}\cdots y_{1,1}\ y_{0,1} \\ \hline s_n\ s_{n-1}\cdots s_1\ \ s_0 \end{array} \qquad y_{n,0} = y_{n,1} = 0$$

These Gray numbers are modified because extra bits $y_{0,0}$ and $y_{0,1}$ have been added to the Gray numbers. The Gray code with the even parity check bits ($y_{0,0}$ and $y_{0,1}$) is known as the modified reflected binary code. The meaning of $y_{0,0}$ and $y_{0,1}$ and the recursive nature of Gray addition is shown in this procedure:

$$y_{0,0} = y_{1,0} \oplus y_{2,0} \oplus y_{3,0} \oplus \cdots \oplus y_{n,0}$$

$$y_{0,1} = y_{1,1} \oplus y_{2,1} \oplus y_{3,1} \oplus \cdots \oplus y_{n,1}$$

$$S_i = E_{i-1}\ F_{i-1} \oplus y_{i,1} \oplus y_{i,0} \qquad E_{-1} = F_{-1} = 0$$

$$E_i = E_{i-1}\ F_{i-1} \oplus y_{i,0} \oplus E_{i-1}$$

$$F_i = E_{i-1}\ F_{i-1} \oplus y_{i,1} \oplus F_{i-1}$$

The problem is to find S_i, E_i and F_i. To find S_i first start with input tape

Input | E_{i-1} | F_{i-1} | y_i,1 | y_i,0 | 0 | 0 ···
S

This is then run through a finite automaton using the following program (a combination of a two digit product and a three term sum modulo two):

1	2	3	4	5	6	7
0 0 3 / 1 0 2	0 0 4 / 1 0 6	0 0 4 / 1 0 4	0 0 5 / 1 0 7	0 0 0 / 1 0 0	0 0 7 / 1 0 5	0 1 0 / 1 0 0

The resulting output tape is then:

| 0 | 0 | 0 | S_i | 0 | 0 ···
F

This same program is used to determine E_i and F_i. The tape must be run through the automaton 3(n+1) times, but the program only requires seven cards.

Binary Subtraction and Multiplication

Because binary operations such as subtraction and multiplication employ binary addition programs, only a description of the procedures is presented.

Binary subtraction is simply modified binary addition with all negative numbers complemented. The complement of a binary number is achieved by replacing all ones in the number by zeros and all zeros by ones and by prefacing the number with a one. Positive numbers are prefaced with a zero. When considering negative numbers the following rule should be followed. If two n bit binary numbers are added and the resulting sum has n+1 bits with a one in the most significant place, replace the one

by a zero and add the binary number one to this result. Negative numbers appear in their complemented form.

Binary multiplication makes use of the fact that multiplication of a binary number by 2^n can be accomplished by simply shifting all of the ones in the number n places to the left. Since a binary number has the form of $\sum_{i=0}^{n} b_i 2^i$, multiplication consists of repeated additions of a shifted multiplicand.

Binary Division

Wilson and Ledley (13) devised an algorithm for binary division which employs addition, subtraction, and decision processes. Consider binary numbers as being composed of strings of ones, strings of zeros, strings of ones including isolated zeros, and strings of zeros including isolated ones. This generates a condensed decimal representation of the binary number. The binary number 0.1111 0000 11011 00100 has a decimal representation as $2^0-2^{-4}+2^{-8}-2^{-11}-2^{-13}+2^{-16}$.

The first step in finding the quotient Q = N/D is to make sure that the denominator D is positive and normalized by making the most significant bit one by a shift of the decimal point. The numerator N should also be positive with N<D and either normalized or with a single zero after the binary point.

The first remainder N' = N-D which is negative is now found. If N' has α_1 zeros to the right of the binary point, then the quotient Q has at least α_1-1 ones to the right of the point. N' is now normalized to N'_N and the second remainder N'' = N'_N + D is formed. N'' is now normalized to N''_N by moving the binary point α_2 bits to the right. If N'' is negative then the

α_1-th bit of the quotient $Q_{0+\alpha_1} = 0$ and the following α_2-1 bits are ones. If N" is positive, then $Q_{0+\alpha_1} = 1$ and the following α_2-1 bits are zeros. This procedure is continued until the quotient has the desired number of bits in it.

As an example of the division algorithm, consider the quotient $Q = \frac{1011011}{1011} = \frac{91}{11}$. Q should first be put into the form:

$$Q = \frac{0.01011011}{0.1011} \cdot 2^4, \text{ where } N < D.$$

```
+ 0.01011011   N
- 0.10110000   D
- 0.01010101   N'      Q1 = 0.?
    ↑

- 0.1010101    N'
+ 0.1011000    D
+ 0.0000011    N"      Q2 = 0.10000?
      ↑

+ 0.11         N'
- 0.1011       D
+ 0.0001       N'      Q3 = 0.10000100?
      ↑
```

Therefore, $Q = Q_3 \cdot 2^4 = 1000.0100 = 8.25$.

Binary Square Root

The square root procedure described is a direct adaptation of the conventional "long-hand" method of square-rooting. Scott (8) gives a recursive relation which describes the procedure. The relation is : $x_{j+1} = 2x_j - a_{j+1}(2A_j + 2^{-(j+1)}a_{j+1})$. The number x_0 whose square root is to be found is written as a binary decimal with an even number of bits to the right of the decimal. $A_j = a_0.a_1a_2\cdots a_j$ is the j-th approximation to the square root. Initially $A_j = A_0 = 0$. Each digit a_{j+1} is either a zero or a one. If the expression $X_{j+1} = 2x_j - (2A_j + 2^{-(j+1)})$ is negative, then $x_{j+1} = 2x_j$, and the bit a_{j+1} of A_j is zero. If X_{j+1} is positive, then $x_{j+1} = X_{j+1}$ and $a_{j+1} = 1$.

As an example consider the square root of 121 = $(0.01111001)^{1/2}$. $(2^8)^{1/2} = x_0^{1/2}\, 2^4$. It is convenient to work the problem in tabular form.

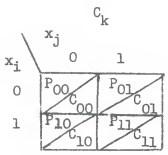
	x_j	x_{j+1}	A_j	$2^{-(j+1)}$
0)	$x_0 = 0.01111001$ $2x_0 = 0.11110010$	$X_1 = 0.01110010$	$A_0 = 0$	$2^{-1} = 0.100$
1)	$x_1 = 0.01110010$ $2x_1 = 0.11100100$	$X_2 = -0.01011100$	$A_1 = 0.1$	$2^{-2} = 0.01$
2)	$x_2 = 0.11100100$ $2x_2 = 1.11001000$	$X_3 = 0.10101000$	$A_2 = 0.10$	$2^{-3} = 0.0010$
3)	$x_3 = 0.10101000$ $2x_3 = 1.01010000$	$X_4 = 0.00000000$	$A_3 = 0.101$	$2^{-4} = 0.0001$
			$A_4 = 0.1011$	

The square root of 121 is $A_4 2^4 = 1011 = 11$ (eleven).

EXTENSIONS OF TURING MACHINES

A Two-square Automaton

A finite automaton which scans two squares of the tape at once is capable of performing all the functions of a regular finite automaton, but requires a fewer number of states. Such an automaton will overprint in the right square of a scanned pair (x_i, x_j) and shift one square to the right according to the typical card below.

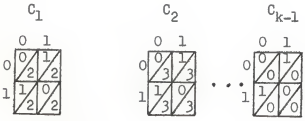


The overprints, P_{ij}, are either zero or one and the C_{ij}'s are the called states.

After a symbol was printed by a finite automaton the machine moved to the right and the overprinted symbol could not be seen again. The program kept track of this overprinted symbol by having the machine go into one state if the symbol was a zero or another state if the symbol was a one.

The output symbol of a two-square finite automaton is part of the input to the next state. Because of this, the machine does not need to keep track of the previously printed symbol by a particular state assignment.

A binary adder modulo two provides a good example of the card reduction feature of the two-square finite automaton. A regular finite automaton requires 2k-1 cards to add k binary bits modulo two. A two-square finite automaton only requires k-1 cards. The binary bits are entered on the tape in the same manner as a regular finite automaton, but the standard starting position is the square to the right of the leftmost bit on the tape. The two-square finite automaton program for the sum of k binary bits modulo two is:

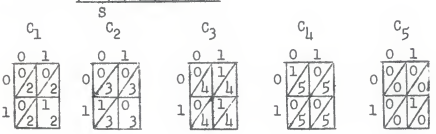


Evaluation of 2-valued Boolean functions of n-variables is an example of an operation which a two-square finite automaton can generally do with fewer cards than a regular finite automaton. For n-variables the regular finite automaton required $2^{n-1}+3$ cards for the standard unreduced form.

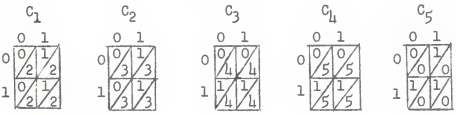
A Boolean function must be expressed in either the maxterm or minterm

canonical form to be evaluated by a two-square finite automaton. Each literal is entered on the tape as it appears in the canonical form from left to right. The number of cards required is 2(k-1) where k is the number of literals in the canonical form.

Two-square finite automata use four types of cards and require two passes to evaluate two-valued Boolean functions. One type of card evaluates sums of literals, another the product of literals, the third is a shifting card and the last transfers a symbol one square to the right. On the first pass the individual minterms of maxterms are evaluated. The next pass forms the sum of minterms or product of maxterms which evaluates the function. The automaton which evaluates the Boolean function f = xyz' + x'y'z is as follows: Input



The output tape is then run through the program below which evaluates the Boolean function f by forming the sum of the two minterms.



The value of f is found in the square which previously had the last literal of f printed on it.

These examples demonstrating the operation of the two-square finite

automaton give but a sampling of its capabilities. For instance, a single card is capable of giving an instruction to go to any of four branches in a flow chart. The next section discusses another aspect of two-square finite automata capabilities.

Two-square Automaton with Cylindrical Tape

Some very interesting results evolve from a one card two-square finite automaton which is given an input sequence on a cylindrical tape having a finite number of squares. The machine always remains in the one state and will print symbols indefinitely. Depending upon the input sequence and the nature of the machine, it might eventually print zeros or ones exclusively or it might set up a pattern of repeating a particular output sequence with a period of a certain number of cycles of the cylindrical tape.

The two-square finite automaton with a cylindrical tape is a model for sequential circuits. This representation of sequential circuits on automata is a good subject for future investigation.

SUMMARY

Procedures for programming finite automata have been exhibited. The basic binary arithmetic operations performed on finite automata required repeated passes. Because of these repeated passes the operator of the machine was required to prepare the tape and keep track of the output data of the machine. The large amount of work performed by the operator was necessary because finite automata have no memory.

Finite automata exist which compute any 2^n-valued Boolean function by representing the Boolean elements by binary elements and then using the

2-valued Boolean operations. In general, finite automata can work with any mathematical system which has a finite set of elements and is closed under the operations of the system. Such a finite automaton can be constructed by setting up an isomorphism between the elements of the system and binary numbers. These machines work like the automata which compute 2-valued Boolean functions. Functional values corresponding to all possible input variables need to be known. Consequently, a system with very many elements requires a finite automaton with a large number of states.

Programming of the general class of Turing machines differs considerably from that of present day digital computers. A digital computer has a complex physical structure which enables the use of rather simple programs. On the other hand, Turing machines quite frequently require complicated programs. The reason is that the "physical structure" of a Turing machine is the program itself. Complexity of an operation on a Turing machine is evidenced by the complexity of its program.

ACKNOWLEDGMENTS

The author expresses appreciation to Dr. Charles A. Halijak, his major advisor, for the suggestion of the thesis topic and guidance in preparing this thesis. Acknowledgment is also due to Professor Tibor Rado of Ohio State University for giving much insight and information on Turing machines.

REFERENCES

(1) Davis, Martin.
Computability and unsolvability. New York: McGraw-Hill, 1958.

(2) Gödel, Kurt.
Über formal unentscheidbare satze der principia mathematica and verwandter system, I. Monatshefte Math. Phys. 38:173-198. 1931.

(3) Lee, C. Y.
Automata and finite automata. BSTJ. 39:1267-1295. 1960.

(4) _____.
Categorizing automata by W-machine programs. J. ACM. 8:384-399. 1961.

(5) Lucal, Harold M.
Arithmetic operations for digital computers using a modified reflected binary code. IRE Trans. of PGEC. 8:449-458. 1959.

(6) Moore, Edward F.
Gedanken experiments on sequential machines. Annals of Math. Studies. 34:129-153. 1956.

(7) Phister, Montgomery.
Logical design of digital computers. New York: John Wiley and Sons. 1959.

(8) Scott, Norman R.
Analog and digital computer technology. New York: McGraw-Hill. 1960.

(9) Shannon, Claude E.
Computers and automata. IRE Proc. 10:1234-1241. 1953.

(10) Turing, Alan M.
On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. Series 2. 42:230-265. 1936-7; with a correction, Ibid. 43:544-546. 1937.

(11) Wang, Hao.
A variant to Turing's theory of computing machines. J.ACM 4:63-92. 1957.

(12) Watanabe, Shigeru.
5-symbol 8-state and 5-symbol 6-state universal Turing machines. J.ACM. 8:476-483. 1961.

(13) Wilson, J. B., and R. S. Ledley.
An algorithm for rapid binary division. IRE Trans. of PGEC. 10:662-670. 1961.

SIMULATION OF ARITHMETIC AND BOOLEAN FUNCTIONS ON TURING MACHINES

by

RICHARD DALE CHELIKOWSKY

B. S., Kansas State University, 1961

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1962

This thesis describes the nature and operation of Turing machines which use a different state representation than that presented in the literature. Turing machines are of interest because of their similarity to digital computers. Any computable function may be evaluated by either a digital computer or a Turing machine.

Turing machines are much simpler in their physical structure than are digital computers. Emphasis falls on programming rather than on machine structure. However, programs for simple operations such as addition and multiplication are quite complicated.

Examples of Turing machines which compute simple functions are given along with a machine that will compute any general recursive function.

Finite automata which are but a subclass of Turing machines are investigated for their computational abilities. Finite automata are limited in their operations because they are not allowed to scan previously computed data. This reduces them to a machine without any memory. Finite automata are found to be particularly suited for the evaluation of Boolean functions. A finite automaton which computes a Boolean function defines that function by its truth table representation. Finite automata can execute binary operations such as addition, multiplication and transformation of Boolean functions.

In an effort to achieve a greater computational scope a variant of an automaton is introduced. This modified finite automaton achieves a reduction in size of most computational programs of the regular automaton. The properties of this automaton variant are not completely investigated, but their possible capabilities are discussed.